TRUST-BY-CONTRACT: MODELLING, ANALYSING AND PREDICTING BEHAVIOUR OF SOFTWARE ARCHITECTURES

Heinz Schmidt
School of Computer Science and Software Engineering
Monash University, Australia

Iman Poernomo
DSTC Pty Ltd
Australia

Ralf Reussner
DSTC Pty Ltd
Australia

Architecture description languages (ADLs) are used to specify high-level, compositional views of a software application. ADL research focuses on software composed of prefabricated parts, so-called software components. *ADLs usually come equipped with rigorous state-transition style semantics, facilitating verification and analysis of specifications. Consequently, ADLs are well suited to configuring distributed and event-based systems.*

However, additional expressive power is required for the description of enterprise software architectures – in particular, those built upon newer middleware, such as implementations of Java's EJB specification, or Microsoft's COM+/.NET. The enterprise requires distributed software solutions that are scalable, business-oriented and mission-critical. We can make progress toward attaining these qualities at various stages of the software development process. In particular, progress at the architectural level can be leveraged through use of an ADL that incorporates trust and dependability analysis.

Also, current industry approaches to enterprise development do not address several important architectural design issues.

The TrustME ADL is designed to meet these requirements, through combining approaches to software architecture specification with rigorous design-by-contract ideas. In this paper, we focus on several aspects of TrustME that facilitate specification and analysis of middleware-based architectures for trusted enterprise computing systems.

1. Introduction

Over the past decade, distributed systems, middleware and software architecture have undergone considerable evolution to meet the needs of the enterprise. The enterprise requires software solutions which are business-oriented, mission-critical (24/7/365), scalable and distributed. It is now generally accepted

that such solutions can be delivered effectively by utilising a component-based middleware, together with rigorous approaches to component-based software engineering (CBSE). Design views in CBSE have several levels of granularity: from a fine grain lines-of-code level to a coarse grain architectural level.

In this paper, we examine approaches to CBSE at the architectural level, to meet the needs of enterprise systems based in middleware.

Examples of newer middleware are those based on Java's EJB specification or Microsoft's COM+/ .NET. Such component-based middlewares and their associated component infrastructures have evolved over the last ten years, mainly with the following industry needs and aims (CBSE Workshop, 2001):

Primary – aimed at sales, market share or cost savings:

1. *Flexibility.* Required by development departments, typically to improve future evolution and extension.
2. *Time to market.* Required by sales departments, with the goal of cutting production time and cost, increasing chances of meeting demands earlier than the competition. Increasingly, CBSE investment is made in a combination of software reuse, version control and product line management (Bosch, 2001).
3. *Cross-department standardisation.* Required by corporate management, primarily to save costs and improve process, and to better meet user/client demand for following best practice and national or international standards (Bosch, 1997, Szyperski, 1998).

Secondary – contributing indirectly to primary aims:

1. *Quality.* Required by users and clients, with a view to certifiability and explicit demonstration of best practice.
2. *Predictability.* Predictability arises from various *extra-functional* measures, including reliability, dependability, robustness, availability, performance and security. Traditionally, these qualities are required by users and developers working in communications, utility, defense and manufacturing areas. More recently, predicability has become important for electronic commerce areas.

We leverage the primary needs of industry through a software engineering process that meets these secondary aims. For large scale enterprise systems, this is not an easy task, and solution providers will profit from some formal specification and design methodology. In particular, secondary requirements of quality and predictability can only be adequately addressed with some level of formal modelling, design and evaluation.

It is in the nature of component-based systems that, while new applications are designed, a majority of components required are pre-built, executable, third-party and proprietary. Consequently, unlike the development of traditional integrated and centralised enterprise software, this availability of executable components in an enterprise system means:

- components cannot be changed easily, so the system will involve adaptation of components.
- components may change independently later, and the system should be adaptable to these changes.
- the software engineer may not have access to component source code or have the knowledge of likely future changes – we must treat them as black- or grey-box.
- components' executables are accessible, so we can test or measure them and develop a realistic model of their behaviour and utility.

- there may be measures of components' performance, so we can predict their future behaviour and performance, modulo changes.
- components may depend on critical network resources with some likelihood of failure – our new application must tolerate such failure.

Compared to traditional integrated systems development, enterprise component-based development must understand the possibility of medium-term change beyond our control, and performance issues. The availability and executability of significant components permits an execution-based approach to prototyping, design and verification, based on interaction with the real environment. However, due to the lack of source code and the risk of depending on potentially changable implementations, interface specifications must capture those benefits of concretely observable black-box behaviour. The resulting separation of component interfaces from component implementation, is one of the hallmarks of software engineering, and has long been accepted in the distributed systems community. But, to better capture black-box behaviour, this separation must include some level of formal interface specification.

There are several specification approaches which are applicable to various facets of component-based enterprise development.

Within the software industry, UML is now firmly established as the object-oriented specification standard. A UML object-oriented model provides an important lower-level view of an enterprise system. For instance, such model is useful in designing basic components of the system. The UML metamodel is based upon the MOF (OMG, 2000a) metamodel. The intention of the MOF is that designs of metamodels for various system views may be given within the same framework, to understand their inter-relationships. Besides UML, an example of such a MOF-based metamodel for the enterprise is the EDOC proposal (OMG, 1999), used for business process modelling.

Formal approaches to architectural views are only recently gaining acceptance in industrial enterprise development. In this paper, we describe a formal approach to combining component technology and architectural description languages for the specification and analysis of functional and extra-functional properties. This permits us to define architectures that meet the secondary and primary goals of enterprise development, through a notion of *trusted* component-based systems.

We proceed as follows. In Section 2, we define our notion of trusted software architectures, and outline several important issues that arise when describing architectures for the enterpise. In Section 3, we introduce TrustME, our architectural description language. Section 4 explains how we adapt design-by-contract notions to TrustME, yielding more trusted architectures. In Section 5, we outline how TrustME enables a faithful modelling of enterprise systems based in newer middleware. Section 6 discusses related work and Section 7 provides concluding remarks.

1.1. Example

Our running example represents a typical subarchitecture that might be used in a larger enterprise system. It involves a simple hotel reservation system, built from three COM+ components – `ReservationSystem`, `ReservationTransfer` and `ReservationBilling`. Upon receiving an event notification from the first component that a hotel reservation is to be made by a user, the second component performes B2B operations with the hotel at which the reservation was made. Upon receiving the same type of event notification, the third component performes billing operations against the user's credit card. When the event is sent out, `ReservationTransfer` and `ReservationBilling` will execute concurrently. However, we require transaction support over both these components, as, if one fails, then calls to either component must be rolled back. This will prevent a user being billed if the hotel, they wish to book at, is full, and will prevent the hotel from accepting a guest if their credit is bad.

2. Software Architecture and Trust

We take the definition of *software architecture* as given in (Ran, 2000, pp. 10-12):

> *Software architecture is a set of concepts and design decisions about the structure and texture of software that must be made prior to concurrent engineering to enable effective satisfaction of architecturally significant explicit functional and quality requirements and implicit requirements of the product family, the problem, and the solution domain.*

What is trust in the context of architecture? The word *trustworthy* is a remarkable mix of objective and subjective notions[1] including *reliable, dependable, faithful, trusty, responsible, credible, believable, loyal, unselfish and true.*

We define trustworthiness (Schmidt, 2001) as *measured and perceived performability* based upon modelling, prediction and evaluation – i.e., based upon an engineering approach to system development. Performability is performance modelling and prediction, combined with systems dependability evaluation. Dependability, in turn, is usually defined as a combination of *reliability, safety, robustness, availability and security* (Littlewood and Strigine, 2000). Performability is achieved through

- proofs and calculations,
- statistical measurement (of extra-functional properties such as reliability, robustness, availability),
- tests (including compliance with standards, style or architectural frameworks),
- enforced process including rigorous planning, reviews and inspections,
- social processes such as public scrutiny and de-facto standardisation (for example, open source development), and
- subjective judgment including testimonies of utility and fitness (Meyer, Mingins and Schmidt, 1998).

A component-based architecture defines how components are configured, deployed and hierarchically organised. We are interested in how trust in individual components can affect the overall trustworthiness of an architecture. That is, we are concerned with the *compositionality* of trust. In (de Roever, Langmaack and Pnueli, 1998), compositional system design is defined as requiring

> *That a program meets its specification should be verified on the basis of specifications of its constituent components only, without additional knowledge of the interior construction of those components.*

Compositionality is one of the hallmarks of CBSE. It requires that a CBSE approach supports reasoning about system properties from properties of the direct system components, without recourse to their implementation details or source code. Ultimately, this means that components come along with some formal or informal specification which we can use in lieu of the components themselves (de Roever, Langmaack and Pnueli, 1998). In configuring components into larger systems using middleware, glue code or architectural composition operators, the architect is also configuring component specifications into a larger specification, permitting the abstraction and reflection of a significant behaviour or property of interest.

In (Meyer, Mingins and Schmidt, 1998), a compositional notion of trusted components was put forward. In the context of component-based architecture, trust in an overall system should be obtained through the system's components, the ways in which they are configured, and associated levels of trust. Much work

[1] *Roget's Thesaurus*, Penguin, 1999.

still needs to be done in understanding this compositional nature of trust in architectures. One promising direction is to leverage current work done in architectural description languages.

Architectural description languages (ADLs) present a compositional, component-oriented view of software in-the-very-large. Most ADLs achieve this through modelling configurations of

- *components*, asynchronous points of computation in a system,
- *connections* between *ports* or *services* of these components, the possible types of communication and interaction within a system, and
- *compound components*, higher-level components composed of interconnected lower-level components (Medvidovic and Taylor, 2000, Kramer et al., 2000).

Because components are asynchronous, and connections are a form of loosely coupled communication, ADLs are well suited to describe concurrent, distributed and/or event-based software at a very high level. Also, most ADLs have well-defined behavioural semantics, so that configurations have a meaning which can be analysed at least in principle to prove, for instance, safety and liveness properties, or to predict performance measures.

In this way, ADLs have a syntax and semantics aimed at defining and analysing distributed configurations and interactions of components. Thus, current ADLs are ideal for describing distributed systems and for describing solutions based on older middleware, such as Microsoft's DCOM and basic CORBA implementations. This is true because, from an architectural perspective, there is little difference between a distributed, older middleware-based system and a concurrent, event-based system.

However, many challenges remain concerning enterprise computing architectures per se, architecture-based component models and trustworthiness in particular (Schmidt, 2000, Littlewood and Strigine, 2000, Lutz, 2000, van Lamsweerde, 2000, Kopetz, 2000, Garlan, 2000). We focus on four issues, now outlined.

2.1. Closer correspondence to middleware component models

There is a wide gap between middleware component and architecture models. The latter have largely developed in research, the former in practice.

It is desirable that architecturally significant middleware elements have accurate and recognisable representation within the ADL. This has several advantages for enterprise development. First, the ADL is made more intuitive, because software engineers can retain a unified understanding how the ADL relates to the application design and implementation. Second, it must be remembered that an ADL is designed to represent compositional structure, and not only a high-level model of distribution and computation for a system. If compositionally significant elements are not present in an ADL, then the ADL does not satisfy this requirement (even if the ADL can model such elements by some intermediate translation).

For instance, the "components" of an ADL should be recognisable as components by programmers familiar with middleware component models. An ADL should incorporate notions such as multiple interfaces, subtyping, substitutability and reconfiguration.

Moreover, the architectural description of a system is not a standalone view. Other system views and design methodologies will be necessary to meet the requirements of the enterprise. By means of a closer correspondence to key mechanisms of such other models, the compositional view represented by an ADL may permit a more seamless integration with these underlying component models.

2.2. Safer connections and interactions between components

There is a clear lack of unified concurrent process models. Concurrent processes, however, are dominant in enterprise systems, which are, by definition, distributed. Concurrent processes are ubiquitous, ranging from high-level business workflow through to networking and multi-threading tasks.

Most ADLs model a component's interface as a set of port or service names and then use connectors between these to constrain interactions between components. Semantically, these correspond to actions or events which might occur through executing a component. Most ADLs define interface elements to possess no signature or specification, so a binding is simply a pair of two such ports. Thus, the task of determining whether ports should be connected is orthogonal to architectural design.

This can be acceptable for defining small-scale architectures, where communication between components is simple (such as Unix scripts which use pipes and filters). Unfortunately, for larger architectures, where complex information is being communicated, this situation leads to increased risk of design failure. This situation is therefore not satisfactory for scalable and mission-critical, i.e., 24/7/365, applications. For the latter, connections must be capable of capturing the significant aspects of complex behavioural interaction patterns, synchronous and asynchronous, between large-scale components of software architectures.

2.3. Trusted compositional systems

In practice, the enterprise developer will need to build trusted systems from trusted *and* untrusted[2] components.[3] This requires formal, architecture-based models of recovery and availability. In natural organisms malfunctions of components occur, are somehow detected, diagnosed and corrected largely without outside intervention, by blocking or removing such components (not the malfunction per se, as we tend to do in software architecture and design). Formal approaches to highly reliable systems based on less reliable components require design for trust and a probabilistic approach to trust prediction and specification. In particular, any such approach must account for

- random physical failure, that occurs at the underlying physical level;
- the lack of access to internals of *black-box components*, making it impractical or theoretically impossible[4] to detect or correct the precise cause of failure;
- a large numbers of external entities on which a component relies, leading to probabilistic models of, for example, external requests, users or resources – this necessitates a probabilistic model of resulting guarantees, dependent on those external entities.

A compositional approach to significant trust properties is needed. It must be possible to reason about system properties based on just the external (or connection/interaction) abstractions of components and without reference to their internal structure. Some promising work has started over the past decade, see for instance (de Roever, Langmaack and Pnueli, 1998). Distributed systems, in particular, require a *linear approach* to compositional reasoning. This is expressed by Dijkstra (Dijkstra, 1969), where the size of the model of the (distributed) environment of a new or changed component must grow no more than linearly with the size of the components:

> *If we ever want to be able to compose really large programs reliably, we need a discipline such that the intellectual effort E (measured in some loose sense) needed to understand a program does not grow more rapidly than proportional to the program length L, and, if the best that we can attain is [...] L^2 we better admit defeat.*

---

[2] less trusted.

[3] similar to van Neumanns "Constructing reliable organisms from unreliable components", in *Collected Works, 1956*.

[4] The component malfunction decreases the trust in its diagnostics capabilities.

2.4. Context-based coupling of components for global computation

The organisation of *widely distributed or global computation*[5] is not well understood. Current middleware mechanisms are largely based on remote procedure call (RPC) which extends static software binding mechanisms to distributed computations. Global computation requires flexible computation and communication mechanisms permitting dynamic reconfiguration of architectures and dynamic connections of architectural components. Appropriate universal models are to be developed.

Promising recent middleware-based models involve interception-based mechanisms for loose dynamic coupling. Examples of this kind of model is *context-based interception* in COM+ or *containers/servers* in implementations of Java's EJB specification. Here, deployed components are conceived as residing within a context (or container) that potentially intercepts and manipulates each call that crosses the context boundary. Contexts provide a pre-programmed scalable, mission-critical infrastructure for housing component instances, enabling the developer to focus on business-oriented design and programming and enforcing container, framework or production-line level constraints useful in guarantees of critical system properties - such as security or availability levels required. A range of contextual services and constraints are offered or imposed, respectively, by the middleware. Examples of such services are security, transactional, diagnostics or recovery services. These services often cut across and add new functionalities to entire collections of components. Consequently, these are difficult to model with current ADLs, where behaviour is generally understood as associated with particular components.

In (Szyperski, 2000), it was argued that there is a strong need to specify context-based systems at the architectural level. Because context-based interception is a valuable feature of newer, enterprise-oriented middleware, it is important that an enterprise-oriented ADL can provide a direct representation of contexts and contextual services. Current ADLs have yet to address this issue.

3. The TrustME Approach

The TrustME[6] ADL was originally described in (Schmidt, 1998), and has been extended in (Ling et al., 1999) and (Schmidt and Reussner, 2000). TrustME is the product of continuing collaborative research conducted in the DSTC and at Monash University. It distinguishes itself from other ADLs, in meeting the demands of enterprise system development outlined above, defining architectures with a compositional notion of trust.

TrustME decomposes a system into hierarchies of *kens*, linked to each other by connections between *gates*, and constrained and configured by *configuration attributes*:

- *Kens* are self-contained, coarse-grain computational entities, potentially hierarchically composed from other kens and subject to distribution. In general, we define a ken as a protection domain with well defined and constrained connections to and from other kens. If a ken contains other kens, then it is called *composite*, else it is called *basic*. For brevity we use the terms *CKen* and *BKen*, respectively.
- *Gates* protect kens, which cannot be directly accessed. All calls, all data communicated and all object migrated to a ken must come through gates. They are the ports for messages and migrating objects between kens. We distinguish *provided* and *required* gates, *PGate* and *RGate*,

[5] The term *"computation"* relates to logical entities such as processes while the term *"computing"* relates to physical devices such as processors.

[6] Acronym for Trusted Component Model and Integration Environment for Distributed Services.

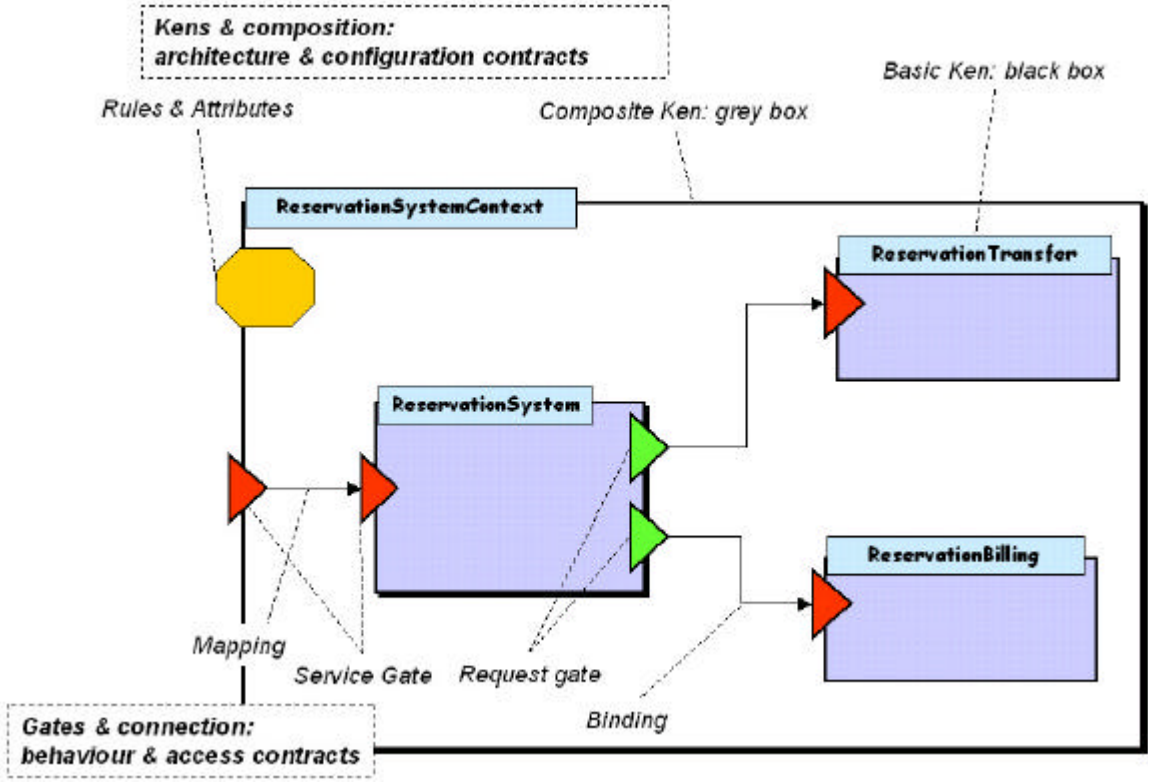

Fig. 1 Representation of our example architecture in terms of kens and gates.

respectively, for brevity. A direct connection from a ken's gate to another ken's gate designates a complex *use* relation, which may be established symmetrically or asymmetrically, statically or dynamically and may be subject to connection constraints.

Configuration attributes define additional properties, constraints and functionality over kens and gates. They can serve as extra-functional property descriptions. For the purposes of modelling enterprise midleware, configuration attributes can be used to define context settings for a context. Just as context settings in COM+ define how the context is to handle deployed components, a configuration attribute provides structural information about what services the ken is meant to provide to the kens it contains. Configuration atttributes are an orthogonal parametrisation of component behaviour in an architecture, because they define constraints and functionality that can cut accross bounderies of components.

Kens and gates are loosely analogous to components and services respectively in Darwin, to components and ports respectively in C2 and ACME, or to processes and ports in MetaH (Medvidovic and Taylor, 2000). However, TrustME differs from these other languages, in that it is augmented with further features to describe middleware-based applications through configuration attributes and other constraints.

Fig. 1 develops a sample architecture for the reservation system example. The outermost ken is a CKen, `ReservationSystemContext,` with a set of configuration attributes. It contains three basic subkens, corresponding to the three components of the example.

3.1. Ken architecture and hierarchy

TrustME employs object-oriented mechanisms for its definitions. A ken is an instance of a *ken class*, and a gate is an instance of a *gate class*. The most general ken class is called `AKen`; the most general gate class `AGate`. These types of classes may be specialised through object-oriented subclassing. This permits reuse of specifications and reuse of glue code. But also, it permits various aspects of system architecture modelling through different subclasses. In this way, kens and gates have a broader range of uses than, say, components in Darwin.

For the purposes of modelling enterprise systems, kens have four important subclasses

- `AKen:` is the most general component class, from which all other kens are derived. It includes behaviour relevant to all kens such as the capability to list its interface objects. It is also a well-defined place for realising corporate standards or project-specific component behaviour such as general diagnostics capabilities, security constraints etc.
- `Bken:` A *basic or black-box component* ken corresponds to a primitive component (for example, a wrapped COTS component). These are the primitive informational and computational resources underlying the construction of a larger system. `BKen` inherits from `AKen`. `BKens` are used to wrap foreign, off-the-shelf, black-box components, lifting to the TrustME context. In our example, we take `ReservationTransfer` and `ReservationBilling` to be instances of `BKen`.
- `CKen:` A *composite component* ken – "C" could also stand for connection, configuration or context – corresponds to a grey-box component. `CKen` inherits from `BKen`, especially black-box operational interface constraints. However some of its architectural details are visible, described by the ken's constituent subkens and thus a `CKen` extends `BKen` behaviour to support grey-box access mainly for configuration or reconfiguration purposes. Typically the number of subkens of a `CKen` is small to moderate. `A CKen` may also be viewed semantically as a composition operator taking certain subkens, instantiating others and interconnecting them according to its grey-box *configuration rules and attributes*. In our example, we assume that `ReservationSystem` is designed by the architect from other components, and so is an instance of `CKen`.
- `DKen:` A *domain component* ken models a large collection of similar components with potential or implicit connectivity and constraints. The interconnect structure is typically established dynamically and – dependent on the type of `DKen` – may include complete pairwise interconnectivity, (shared) bus connectivity, element-wise (index bijection) interconnection of two subarrays of subkens for parallel computations. `DKen` inherits from `CKen`. `DKen` derives its power from its dynamic interconnection capabilities combined with `CKen` configuration rules and attributes. In our example, `ReservationSystemContext` represents a COM+ context in which, for instance, transactional settings for contained components are specified. So, from the perspective of our ADL, this ken is an instance of `DKen`, with the ken's rules and attributes defined to model possible COM+ settings.

New ken subclasses can be devised to model other computational entities: for instance, assemblies in .NET, or networks of machines.

The ken hierarchy is a strict composition or tree hierarchy. This means two different kens do not share subkens. Consequently partitions of kens can be mapped naturally to separately distributed subsystems.

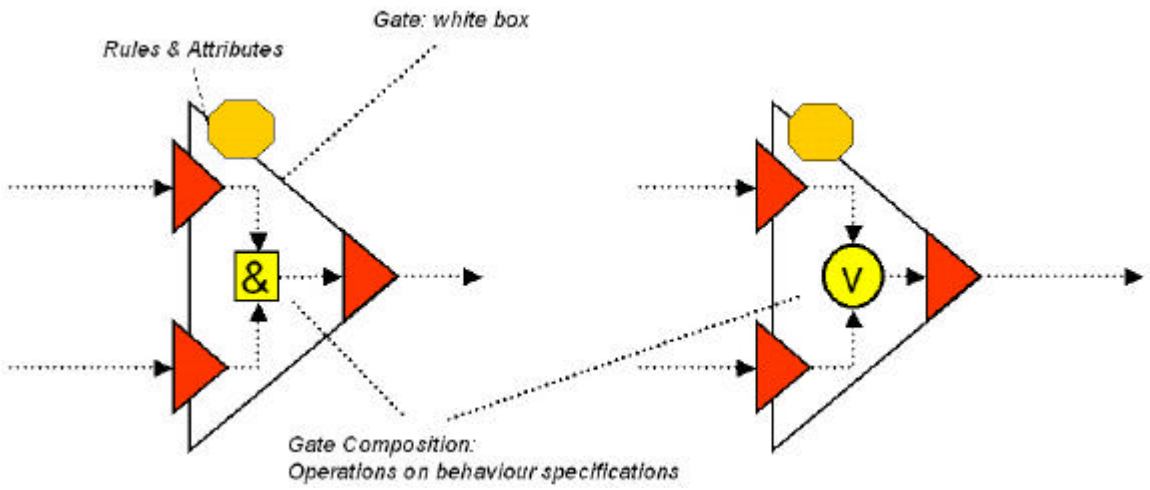

Fig. 2 Gate algebra operators compose state machines or Petri nets.

3.2. Gates and gate connectivity

Gates possess a richer language than the interfaces of components in other ADLs (for instance, ports of Darwin).

- Gates denote *interfaces* to kens, consisting of service signatures and, where appropriate, additional specifications of the behaviour, called *rules*.
- Gates are instances of gate classes. Consequently, gates are analogous to interface objects, adaptors or wrappers in programming.
- Gates protect existing functionality and permit substitution between existing kens in a configuration, subject to compatibility checks over gates.

Gates can be used by kens in two possible ways: as *required* gates, describing interfaces of other kens that the ken requires, and as *provided* gates, interfaces that the ken provides to other kens.

- Required gates (`RGate`) are distinguished from provided gates (`PGate`) which provide the actual service methods or implementations. Nevertheless `RGates` are more than just abstract interfaces to those services. As discussed above, distributed systems compositions requires adaptation and glueing predominantly at the caller site due to the lack of control over foreign services.
- The distinction of `RGates` and `PGates` introduces a direction of *use*. The direction of use is represented visually in TrustME by triangles: inward triangles denote provided gates, outward triangles denote required gates for a ken. Note, however, that gates are complex entities and information may flow back and forth along connections between gates. So, in practice, the direction of use should be thought of as dominant direction of message flow.

Communication between kens is denoted by connections between gates:

- TrustME distinguishes connection *mapping* from *binding* (see Fig. 1). Mappings are “vertical”

connections between two `RGates` or two `PGates` at different hierarchical levels. For example the `RGate` of a `CKen` may be mapped to the `RGate` of one of its subkens. A binding is a "horizontal" connection between an `RGate` (of one subken) and an (of another subken) in the same `CKen`.

- Gates of different kinds may be connected depending on the properties of a connector. Compatibility may be enforced according to different policies including subtyping (the provided interface is a subtype of the required interface) or adaptation (connections protocols include translations).

Kens may have several required and/or provided gate objects permitting them to separate and control differently a variety of communication and interaction capabilities, each associated with a complex gate type (signature and rules) and a gate instance (adapter/connector object).

Gate behaviour is defined by an interface, consisting of service signatures and *rules*. Rules may be

- logical assertions, descriptions of how methods of a gate's signature are meant to respond to being called,
- state machines, descriptions of the order in which the methods of a gate's signature should be called (their *protocol*), and/or
- Petri nets, descriptions of coarser-grain concurrent behaviour, such as business-process workflow.

Many different interface models for software components exist (see e.g., (Krämer, 1998, Vallecillo et al., 1999, Reussner, 2001b)). The simplest interface model for components describe provided and required interfaces by lists of service-signatures (OMG, 2001). In this case, the required interface asks the environment for certain services and the provides interface names the signatures of offered services. (A signature includes name, type of return-value, and parameter types). Signature-list based interface models insufficiently document the deployment of a component. A signature-list based interface does not provide enough information to detect many common errors when checking the interoperability of components or substituting components during reconfiguration.

In the TrustME project we have developed the foundations for a gate algebra based on composition of state machines and semi-automatic adaptation. An extension of this algebra for Petri nets is work in progress. Fig. 2 shows two simple gate compositions using a product and union operator on state machines. In general, complex compositions are encapsulated in hierarchical gate definitions themselves. Similar to kens, such gate definitions may include rules and attributes to realise global constraints (relative to the subgates of such a definition).

Rules provide a semantics for our architectures: they tell us how gates behave, and, by the nature of our architectural compositions, how configurations of kens behave. After the design of an architecture, these semantics are open to analysis and verification, to provide a measure of overall functional trustworthiness. Such a stage, occuring after design, is common to methodologies that use ADLs. Our ADL differs from others, in that its multiple kinds of rules permit several semantic views of an architecture, and hence a range of possible understandings of trust. This is an advantage, because it provides several complementary views of the behaviour of an architecture, enabling the identification of a variety of problems in a configuration of components.

Also, in contrast to other ADLs, by associating rules with gates of kens, we enable the construction of trusted architectures at the design stage, through an adaptation of design-by-contract approaches.

4. Contracts for Trusted Architectures

Within the domain of programming language design, there are convincing arguments for adding further semantic annotations to component interfaces, to make interface usage safer. Essentially, it is argued

that an interface signature alone does not tell us *what* the interface methods are supposed to do, and that this information is necessary to use the interface safely.

These arguments carry over to the ADL world. One popular means of providing semantic annotations for object-oriented designs is *design-by-contract* (Meyer, 1992, Meyer, 1997). In (Schmidt, 1998) and (Schmidt and Reussner, 2000), we adapted design-by-contract to the architectural design level, through use of a gate's interface rules.

The design-by-contract principle was originally defined for class methods (or, in general for functions), based on the pre- and post-condition approach to functional specification given by Hoare (Hoare, 1969). According to (Meyer, 1997, p. 342) a contract between the client and the supplier consists of two obligations:

- the client must satisfy the pre-condition of the supplier.
- the supplier has to fulfill its post-condition, if the precondition was met by the client.

Each of the above obligations can be seen as the benefit for the other party. (The client can assume the postcondition holds, if the precondition is fulfilled. The supplier can assume the precondition holds). A contract specifies that, if the client fulfills the precondition of the supplier, then supplier will fulfill its postcondition.

We define an analogous, generalised notion of contracts for *components*, rather than for single methods.

We consider the user of a ken as the client, and the ken as the supplier of services. The client uses the ken through a required gate. The ken supplies services through a provided gate. By associating *rules* with required and provided gates of a ken, we define properties that must be satisfied by the user of the ken or by the ken as a supplier of services, respectively. In this way, a kens's required gate (an instance of `RGate`) is analogous to a precondition in (Meyer, 1997). Similarily, a ken's provided gate (an instance of `PGate`) is analogous to a postcondition. If the user of a component fulfills the component's required gate's rules (i.e., offering the right environment) the component will offer its services as described by the rules of the provided gate.

So, a contract for a ken involves its gates' interfaces. In our notion of design-by-contract, correct contractual *use* of kens occurs when we connect kens through gates that have *compatibile* rules. We now outline this.

4.1. Contracts for compatible replacement and architectural stability

In the TrustME ADL, we define compatibility between gates in terms of subtyping and partial conformance. We distinguish different forms of compatibility (see also Fig. 3):

- *connection conformance*: the target gate (mapped or bound to) has at least the required functionality and satisfies all required rules (see below). Intuitively the target gate is a subtype (subject to additional functionality and rules) of the using gate. For example the requested services named `A` and `B` are supported by the top left gate in Fig. 3.
- *strong ken conformance*: all `RGates` are connected to conforming gates and all connected `PGates` conform. For example in Fig. 3, `S` may remain unconnected without compromising strong conformance[7] .
- *restricted ken conformance* permits nonconforming and unconnected `RGates` if we can prove compositionally (i.e. by analysis of the ken specification alone) that the missing behaviour is not

[7]This is only true if we assume independence of the two provided gates. Synchronised state machines or Petri nets would have to be modelled using a single composite gate.

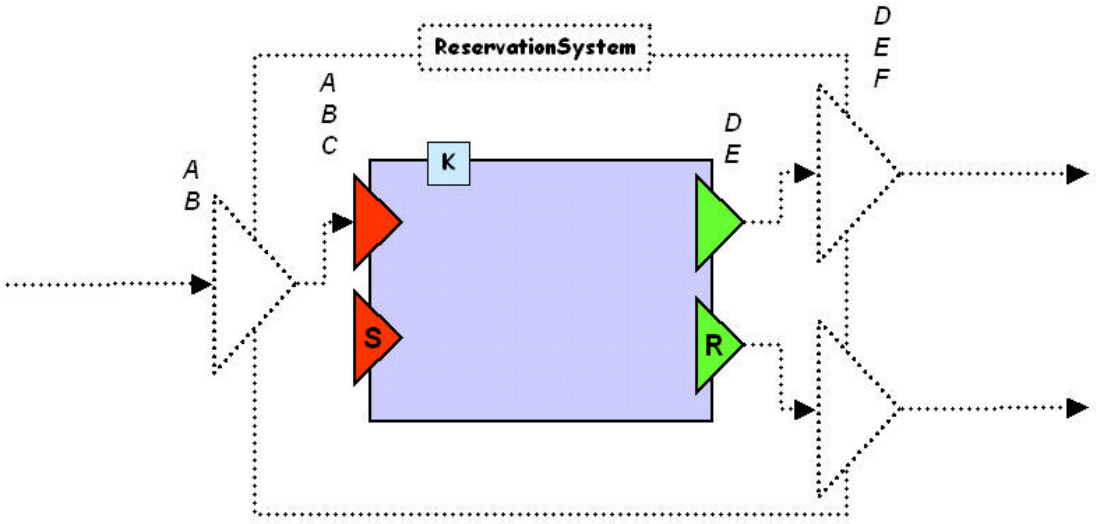

Fig. 3 Compatible replacement.

required (used) given the restriction to a subset of the `PGates` in the replacement. For example, the required gate R might acceptably remain unconnected if its services are exclusively reached from `S` through `K`.

Our notion of compositionality must encompass substitutability of components. In turn, we make substitutability more trustworthy through checking that the replacement is compatible to the substituted. This is a stronger notion of substitutability than that of other ADLs: for instance, in Darwin, the architect could substitute one component for another provided that port interfaces have the same names. In (Schmidt, 1998) we have shown that compatible replacement based on strong conformance guarantees architectural stability. This means, if we assume a correct distributed system (satisfying all rules), then the system resulting from a compatible replacement is correct (satisfies those rules).

In (Reussner, 2001b, Reussner, 2001c) we demonstrated, that restricted ken conformance can be handled efficiently by an extension of contracts. Given a specific reuse-context a so-called parameterised contract can compute the `RGates` describing the external functionality required in this reuse context. Parameterised contracts are an invertible mapping between the functionality offered by a ken (as given in the `PGates`) and the functionality required by the ken (as specified by the `RGates`). This allows a ken to compute its offered functionality as a function of the gates bound to its `RGates`. And, vice versa, the ken can compute its required functionality as a function of the environment it provides services to. This concept increases the reusability of the ken in that it permits automatic adaptations for the context of reuse.

While this result is a strong theoretical argument for the compositionality underlying our extended principles of design-by-contract, practically its use is limited to designers of software engineering methods rather than system implementors. The implied notion of "correctness" in the result depends on the kind of rules one chooses to specify the properties of the system and on a certain completeness of the specification. For distributed systems however specifications remain partial and limited to critical aspects of the system for feasibility reasons.

The above stability result is based on design-by-contract and a logical interpretation of rules as formulae satisfied in states of a state transition systems underlying state machines or Petri nets, as summarised in Fig. 4.

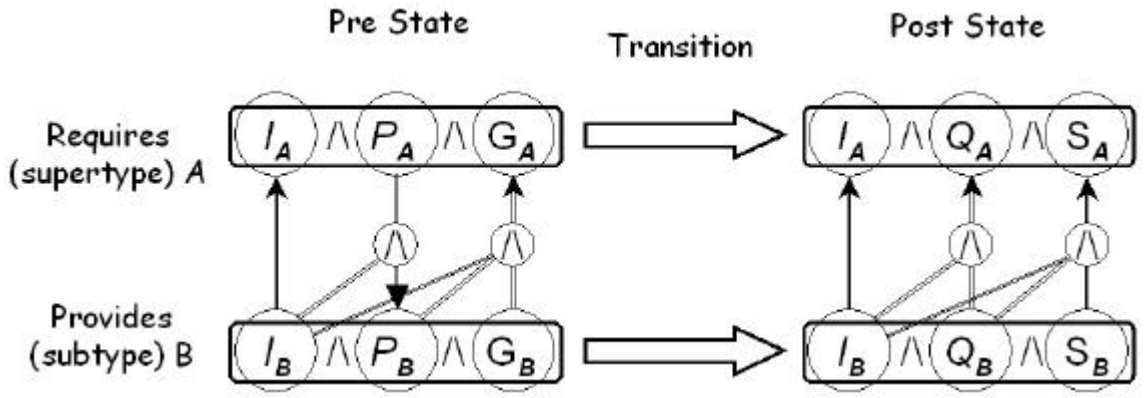

Fig. 4 Compatibility is based on transition-wise conformance modulo context-based adaptations of transitions (see interception rules).

In the context of composition the properties of the replaced component interface are known *and expected* in terms of pre- and postconditions to services and requests on a per-service (method) basis. The upper horizontal transition shows this as the "supertype" (more general than the many possible implementations or replacments). The "subtype" interface (of the more specific) replacement conforms service-wise and logical-rule wise, i.e., in terms of precondition (P), postcondition (Q), invariant (I), guards (G) and progress conditions (S) as indicated in Fig. 4 by implication arrows. Conformance and hence the implied compatibility is transitive, that is, such diagrams can be cascaded horizontally along transition chains (abstract traces) and vertically along replacement or subtyping chains. Each horizontal step in such a reasoning lattice represents step pre- and postconditions as known from Hoare logic. In the presence of encapsulated components or objects, it is not only stylistically desirable to separate the invariant and associate it to the object once, rather than to the pre- and post-conditions of all methods. As seen in Fig. 4 it is rather necessary, because the invariant is treated differently from the remaining precondition to achieve conformance in vertical chains and ultimately architectural stability. Furthermore, in the presence of subtyping, guards and progress conditions, on which modelling and analysis of synchronisation is based, must behave covariantly relative to subtyping, and hence differ from preconditions, which behave contravariantly, in order for local compatibility to suffice for global stability. In contrast, progress conditions and postconditions could be modelled by the same mechanism in theory.

4.2. Finite state machines for protocol rules

The TrustME approach incorporates various types of formalisms for giving interface rules. Here, we show our ADL specifies protocol rules, using finite state machines. Kens have several different services to offer. These services can be called in different orders (i.e., *call sequences*). Protocols arise due to the fact, that usually not all of these call sequences are supported by a ken. For example, a file must be opened *before* one can read or change it. Likewise, it must be closed after accessing it.

The protocol of a provided gate defines the possible sequences of interface service calls. A *valid* call sequence for an interface is a call sequence which is actually supported. For example, the interface to a ken representing a file management component, might include `open-read-close` as a valid call sequence, while `read-open` is not included. Analogously, the protocol of a required gate is a superset of the calls sequences, by which the ken calls external services. In both cases, a protocol is considered as a set of sequences.

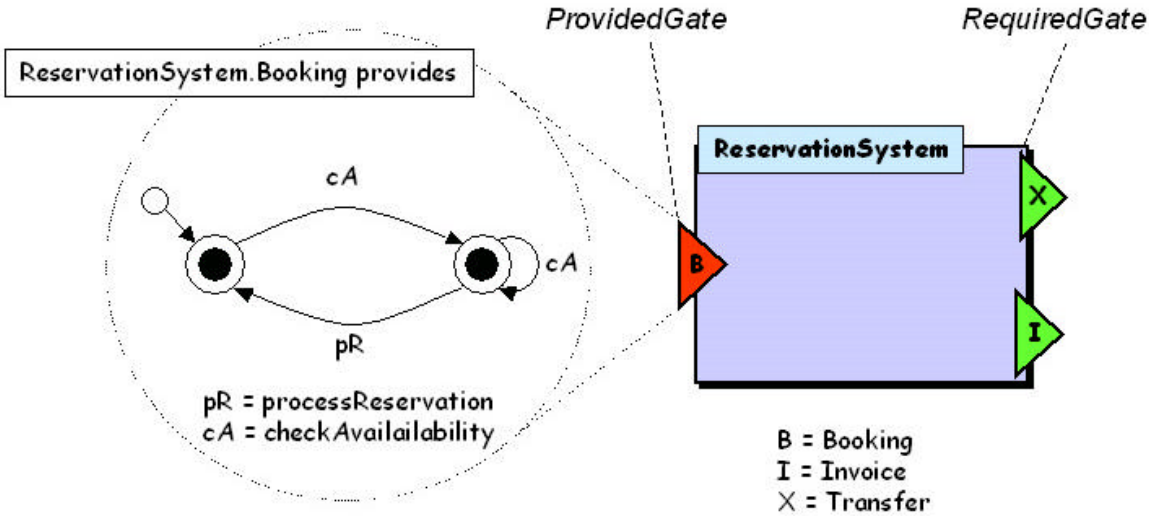

Fig. 5 A finite state machine that defines a protocol for the provided gate of the `ReservationSystem` ken.

Finite state machines (FSMs) are a well-known notation for protocol specification (Nierstrasz, 1993, Yellin and Strom, 1997, Holtzmann, 1991). A state machine specification of protocols enables representation of protocols in a compact and precise manner, and permits automatic formal analysis of protocols.

FSMs (Kleene, 1956) comprise a finite set of states, and transitions between them. To model call sequences, we define the services that are callable for each possible state. Often, such a service call changes the state of the state machine, enabling some services to be callable, and disabling other services that may have been callable in the previous state. Such a change of states is called a *(state) transition*, and, in our work, denotes a service call. A finite state machine, as a definition of possible state transitions, describes a protocol for possible call sequences.

A *state transition diagram* is used to represent a finite state machine. Fig. 5 shows a FSM that defines a protocol rule for the provided gate of the `ReservationSystem` ken. States are denoted by circles, transitions by arrows. There is one designated state, in which the ken is after its creation (i.e., the *start state*). From this start state, all valid call sequences start. A call sequence is only valid if it leads the FSM into a *final state*. These final states are denoted by black circles within the states. In our example all states are final states. This FSM specifies that we should make one or more calls to the `checkAvailability` service prior to calling the `processReservation` service (one cannot make a reservation without checking the availability before). So, a valid call sequence is `checkAvailability-processReservation`.

We describe the protocols of a `RGate` in a similar way. Fig. 6 shows a (trivial) required protocol for the `Transfer` required gate: it specifies that the ken needs multiple calls to the `makeReservation` service.

We also use FSMs to describe the required protocol for particular services of a provided gate interface. For instance, the FSM of Fig. 7 describes the possible sequence of outgoing calls that may result from calling the service `processReservation`. In this case, we have constrained the outgoing call sequence to `makeReservation` followed by `makeInvoice`. In general, there could be mutiple possibilities.

If we know the call protocols for all provided services, we can automatically construct a detailed required protocol for the ken (Reussner and Heuzeroth, 1999).

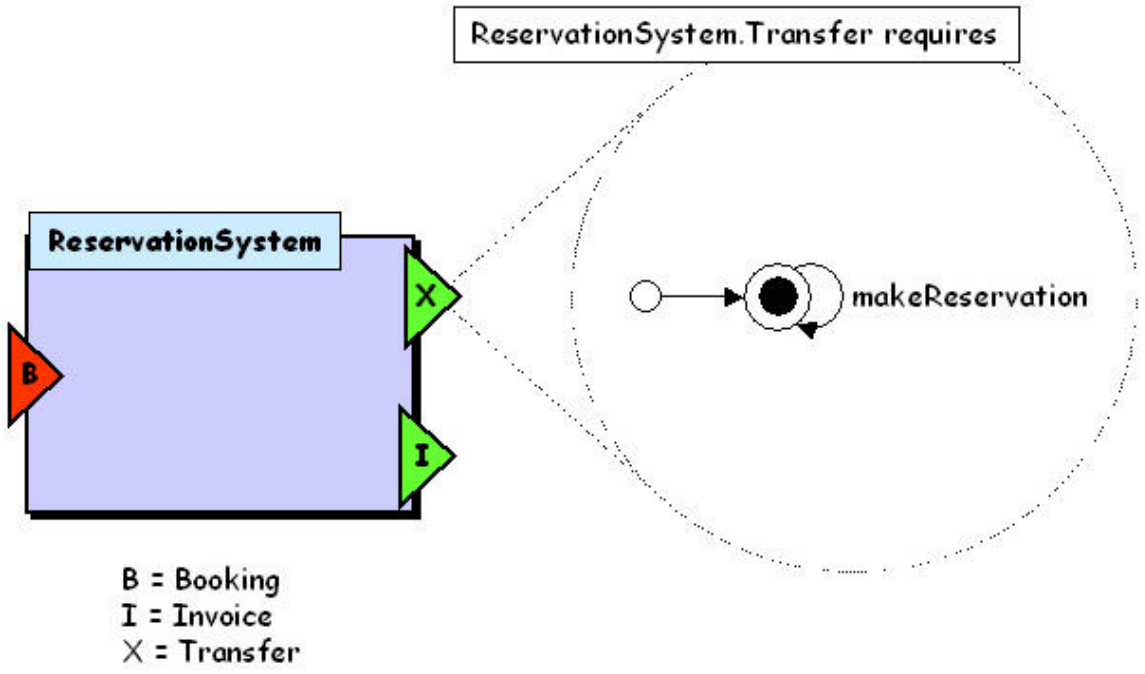

Fig. 6 Required protocol for the Transfer gate.

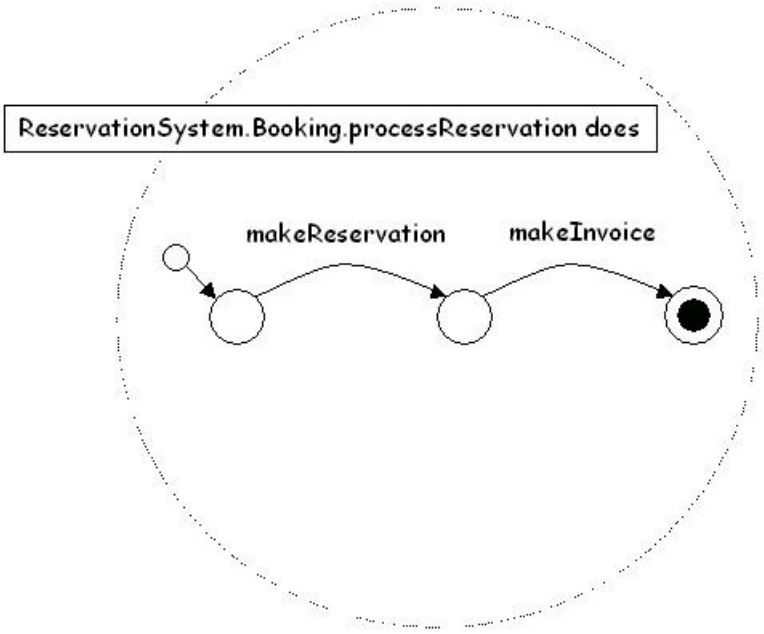

Fig. 7 The possible sequence of outgoing calls that may result from calling the service `processReservation`.

Table 1 Different rules for different purposes.

	Execution		
Aspect	**Pre**	**All**	**Post**
Logical	Precondition	Invariant	Postcondition
Synchronisation	Guard condition	Protection	Progress condition
Interception	Prelude	Interlude	Postlude

4.3. Different rules for different aspects of trust

We can categorise the kinds of rules provided by TrustME according to Table 1.
Our rules enable stabler composition and safer connections in the following ways:

- Logical (side-effect-free) rules serve interface specification, according to the principles of design-by-contract. Such rules specify properties of observable states before and after method execution (pre- and post- conditions) or invariants required to hold in all externally observable states (after creation or service method execution including all PostProvides interception chains).
- Synchronisation rules exclude unsafe states such as overflows, contact, resource contention, blocking etc. They are logical formulae however causing or asserting the delay of actions or locking of components during safety-critical stages.
- Interception rules define transformations of message flows between kens and their gates to establish context-based global policies, modifications and adaptations. This increases the reusability of a black-box component as-is.

Invariant logical rules associated to particular kens represent configuration invariants that must be met after all reconfiguration actions and hence are invariant during the operation of these kens. For this purpose we distinguish (re-)configuration actions from computational operations in the set of methods supported by a ken.

Invariant rules associated to particular gates represent flow or communication invariants - typically safety or performance conditions of the corresponding state machines.

Rules may also capture extra-functional properties. To this end the TrustME ADL introduces cost attributes defined as measures on event transition systems (Schmidt and Zimmermann, 1994). For instance the difference in the number of ReservationBooking requests and completed ReservationBooking services may not exceed a permitted contingency – a measure of event occurences expressing the safety of the booking progress. If we add failure and success events representing system failure (transition to an explicit error state) and progress to any successor state, we may sample event sequences and measure the ratio of failure samples to success samples for selected gates or kens resulting in their failure rate. Using design-by-contract principles – a logical formula can now capture requirements to reliability or availability.

5. Modelling Context-Based Interception

In essence, middleware provides the ability for contexts to manage the interaction between components, without the need for the programmer to write management code. Particularly, this permits addition of

Table 2 Events associated to method M.

	Event		
Gate	**Pre M**	**Default M**	**Post M**
Provided	PreProvides	DefaultProvides	PostProvides
Required	PreRequires	DefaultRequires	PostRequires

functionality to component deployments, by means of context *services*. If a service is used by a component, the details of how this is achieved are largely hidden from the component developer: it is the middleware's responsibility to provide the service to the component. Thus, context-based interception makes scalable and mission-critical development safer, through delegating responsibility away from the developer.

5.1. General approach

Usual services provided by modern middleware include security management, persistence, message queueing and transaction management.

Required gates may associate rules to request events (PreRequires, DefaultRequires, PostRequires) for their methods (cf. Requires row in Table 2). For the purpose of composition, connections are considered as pairs of gates including their associated connection protocols defined by constraint rules and attributes. These constraints are associated to the conceptual events depicted in Table 2.

Similarly provided gates optionally associate PreProvides, DefaultProvides and PostProvides rules for methods (cf. Provided row in Table 2). Such rules may be associated selectively to specific methods or to `Any` method of the gate. For example a `DefaultRequiresAny` rule might define an adapter handling unprovided method calls for example, for realising a standard usage fault protocol.

These rules are inherited and can be extended, i.e., wrapped in a subclass through preceding the implicitly inherited rule by another Pre rule and following it with another Post rule, or replacing Default rules thus composing a natural interception rule chain for subclasses to inherit.

`CKens` (incl. their `DKens` subclasses) may impose connection rules by adding PostRequires and PreProvides rules to bindings (connecting subkens, see Fig. 9). These are "inserted" in the corresponding connections like protocols in a protocol stack, and may apply specifically to a particular method, a gate type, gate instance or apply to any method connection. For example a `DKen` might add a PostRequiresAny rule encrypting requests and a PreProvidesAny rule decrypting them thus realising a level of secure tunneling in its domain.

PreProvides rules may also be associated to `PGate` mappings and PostRequires rules to mappings thus forming interception rule chains along chains of mappings (Fig. 8).

The object-oriented nature of TrustME entails that component kens are closer to components of object-oriented middleware, because notions such as substitutability and subtyping are available to the designer. The ken and gate hierarchy defines interception rule chains following compositionally the ken architectural hierarchy and the extension hierarchy defined by inheritance among ken and gate classes.

The resulting connections defined by these rules can thus model context-based interception of the sort used in COM+ and its successor .NET – yet with well-defined formal compositional semantics. Configuration rules may be parameterised by *configuration attributes*. These are entities of TrustME that model configuration settings for a context. Just as context settings in COM+ define how the context is to handle deployed components, a configuration attribute provides structural information about what

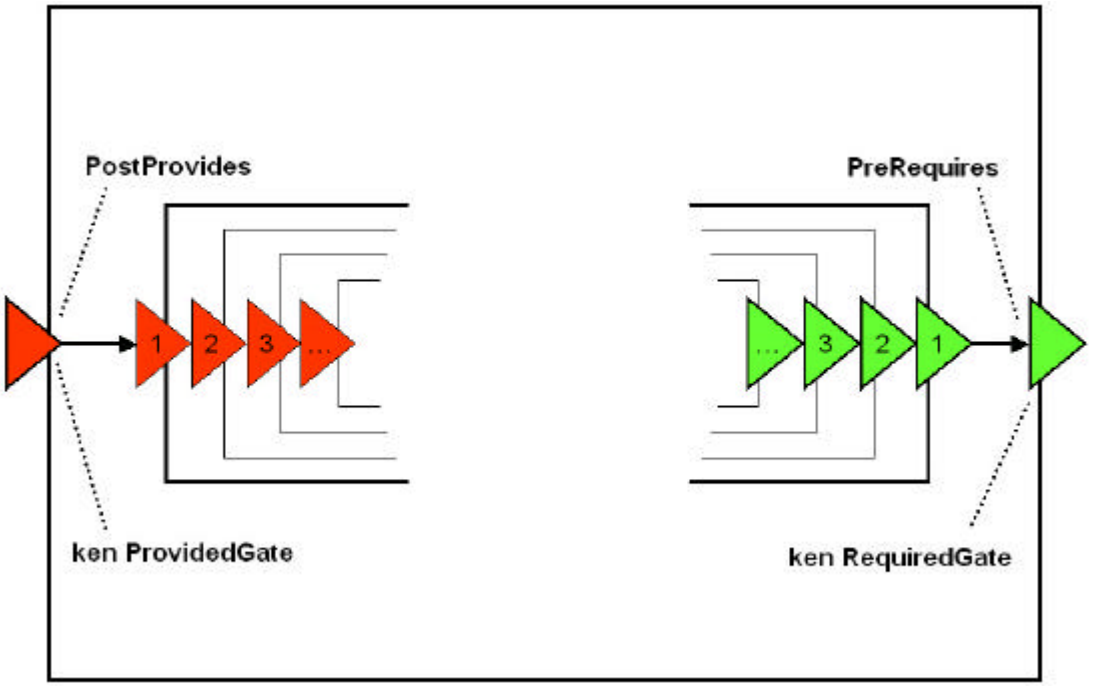

Fig. 8 Gate mapping contracts.

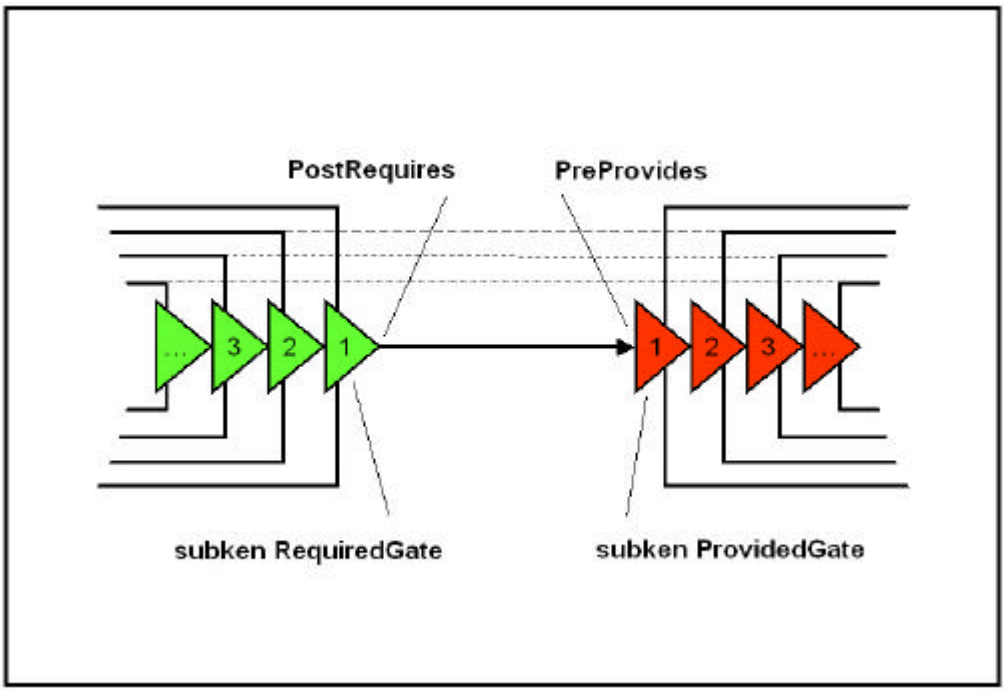

Fig. 9 Gate binding contracts.

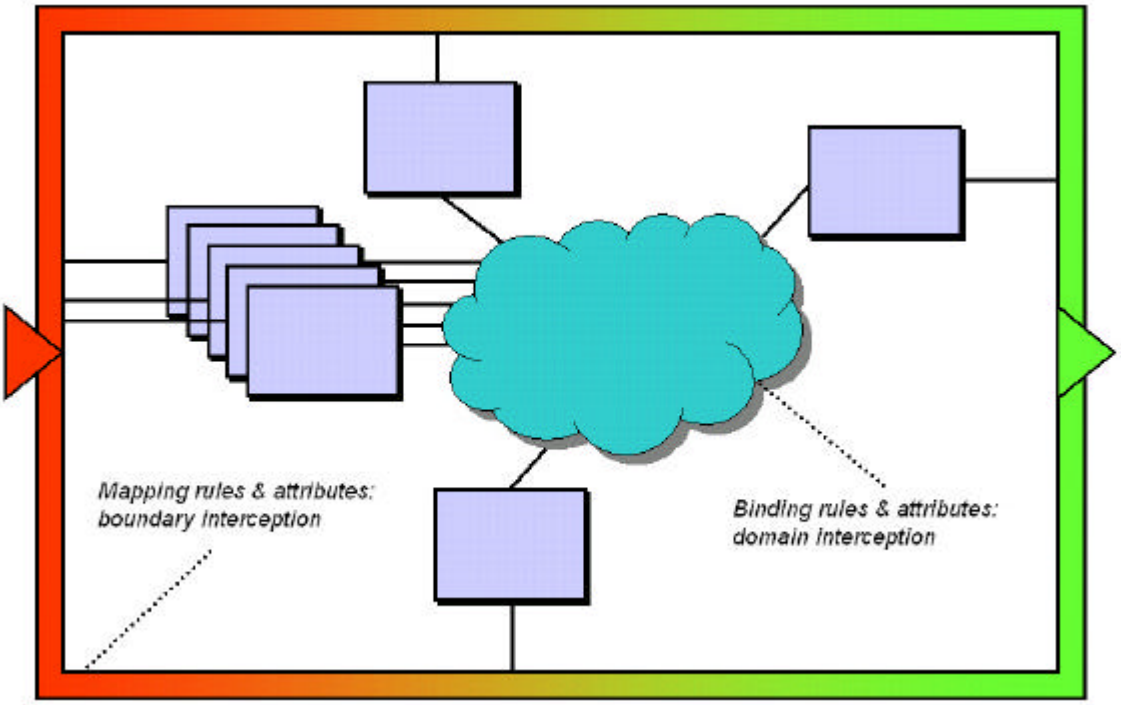

Fig. 10 Context-based interception.

services the context ken is meant to provide to deployed subkens. Semantically, a configuration attribute affects the dynamic behaviour of the subken interaction.

These features enable us to retain a close correspondence with middleware component models - yet blended cleanly with design-by-contract principles lifted to the level of architectural design. In particular, multiple interfaces for a middleware component are simply modelled by a component ken with multiple gates. This is difficult to achieve in many other ADLs.

5.2 Configuration attributes for COM+ transaction services

A transactional COM+ architecture involves COM+ components, contexts and transaction settings. We require representation of these three entities. COM+ offers five possible transaction settings, which we model with five transaction configuration attributes. To model COM+ components and COM+ contexts respectively, we will define subclasses of `BKen` and `DKen` respectively. These subclasses extend the behaviour of their parent classes, but have additional properties needed to reflect a COM+ setting.

In modelling transaction settings, our configuration attributes and subkens need to satisfy syntactic and semantic requirements:

- *Reflection of epistemology.* Important COM+ architectural conventions should be reflected in syntax. The use of transaction settings in implementations should be reflected by the way transaction configuration attributes are used in our ADL. In particular, the encapsulation of attributes within kens in an architecture should model the way in which settings are associated with components in a COM+ architecture. Transaction settings may be pre-set for individual components in their code, or may be set during deployment within a context. We reflect these two possibili-

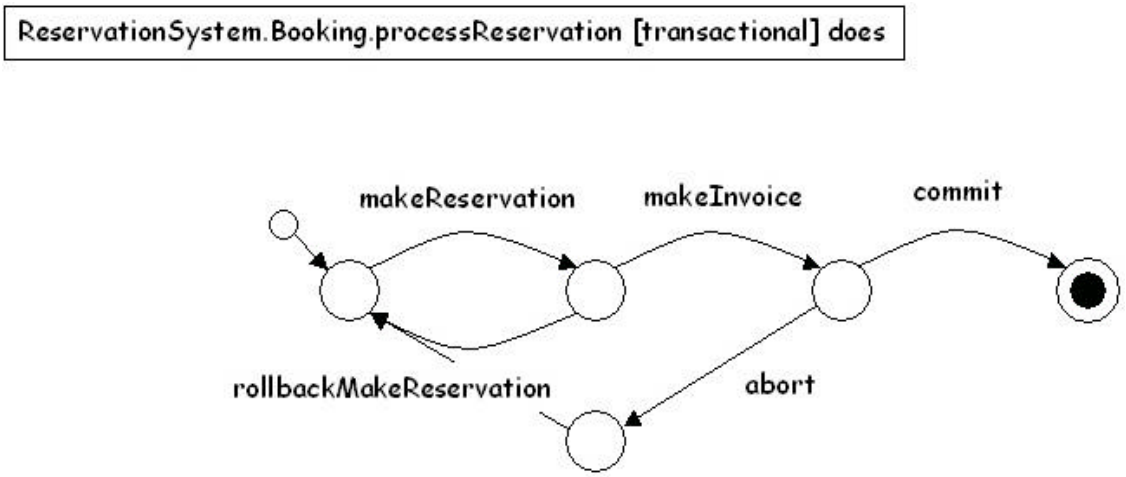

Fig. 11 The required protocol of the service processReservation, transformed to incorporate transactional behaviour.

ties in our ADL by permitting transaction configuration attributes to be encapsulated within ind vidual component kens.

- *Representation of ontology*. The behaviour of a COM+ architecture will depend on how transaction settings are associated with its components. Our configuration attributes reflect this, by affecting the overall semantics of an architecture. To achieve this, we describe how transaction attributes affect the dynamic (FSM) behaviour for component and context kens, and how the contract assertions for component and context kens are affected.

The presence of a transaction attribute in a `DKen` corresponds to the association of a transaction setting with a component, residing within a context.

In our example, the addition of a transactional configuration attribute will change a range of architectural properties. For instance, the required protocol of the service `processReservation` will be transformed to incorporate transitions corresponding to abort, commit and rollback operations of a transaction. This is depicted in Fig. 11 below. The architect need not specify this full protocol. Instead, he/she only need define a basic protocol – in this case, the one of Fig. 7 – and add a transactional attribute to the overall domain ken.

6. Related Work

Notions of conformance and compatibility have been defined in greater detail for objects in (Schmidt and Zimmermann 1994, Frick et al., 1996, Frick et al., 2000) and for components in (Schmidt 98, Schmidt and Reussner, 2000).

Component models of wide use are given by current industrial middleware platforms (i.e., the Common Object Request Broker Architecture (CORBA) from the Object Management Group (OMG, 2001), Enterprise Java Beans (EJB) from Sun Microsystems (EJB, 2001) and the Microsft's Common Object Model COM+/.NET (.NET, 2001). In components of these platforms component interfaces are modelled as signature lists. With the exception of CORBA, required interfaces are not specified. In all models rules for specifying valid call sequences are not given. As argued in this paper missing required interfaces

hinder interoperability checks. Missing rules decrease the benefit of interoperability and substitutability checks substantially, because many common errors remain undetected.

These drawbacks of commercial component models and their precursors in research labs gave rise to interface definitions including rules for behavioural specifications. These rules have been expressed in different notations, each having specific advantages and drawbacks.

Logical predicates: Predicates are the first and most powerful approach for specifying pre- and postconditions. First worked out as a method of program verification (Hoare, 1969) it has also be used to enhance interfaces of methods (Meyer, 1992, Liskov and Wing, 1994) and interfaces of components (Zaremski and Wing, 1997). It is far more general than only describing protocol constraints. Unfortunately, its generality and expressiveness, is also its major drawback, when checking properties. Interoperability and substitutability of components with such interface specifications are undecidable in the general case. Due to that, several restricted versions of a general predicate logic approach have been developed. For modelling protocol temporal logic approaches are of most interest (Manna and Pnueli, 1992). Specifically the Linear Timed Logic (LTL) (van Leeuwen, 1999, p. 995) is of practical use, since interoperability and substitution are checkable by a model checker. In (Han, 2000) a temporal logic based annotational syntax for component interface protocol specification is presented.

Process algebras: Canal et.al. in (Vallecillo et al., 1999) enhanced interfaces with the p-calculus, a special calculus to describe communication processes (Milner, 1980, Milner, 1998). This powerful approach can model protocol information easily but suffers the same problem as some of the approaches that use logical predicates: checking protocol consistency before running the program is too expensive for practical purposes.

Petri nets: Petri nets (Petri, 1962) are a powerful and widely used modelling technique for concurrent processes. A broad variety of variants and algorithms for checking various properties exist - e.g., (Reisig, 1985, Reisig, 1992). In (van der Aalst et al., 2000), Petri nets are used for component models within software architectures. In (Vallecillo et al., 1999), van Rein describes his system Paul which is based internally on a kind of Petri net to model workflows. With this system protocol consistency can be checked.

While efficient algorithms exist for some Petri net models to check global properties (such as liveness, absence of deadlocks, performance guarantees etc.) other properties which are important for architectural system configuration (like interoperability and substitutability checks) cannot be checked in general. Some restricted classes of Petri nets can be translated into automata, and, hence, can make use of their benefits.

Automata based approaches: The use of finite state machines to model protocols and to check their compatibility is well known from the telecommunication and distributed systems communities, e.g., (Krämer and Schmidt, 1987, Holzmann, 1991). Also for modelling object behaviour and specifying and automating test sequences finite state machines are deployed successfully (Bochmann et al., 1982, Harel, 1987). Nierstrasz proposes their use to model the type of an object (Nierstrasz, 1993): in our context we can use the terms “type” and “interface” synonymously, since, as Nierstrasz points out correctly, a type defines the applicability of the typed entity. In our approach we also use interface information to apply kens. But Nierstrasz uses finite state machines only to describe the provided interface. Extending this approach, Yellin and Strom model in one finite state machine both, provided- and required interfaces (Yellin and Strom, 1994). Primarily, they use this interface information to create adaptors automatically (Yellin and Strom, 1997). In (Reussner and Heuzeroth, 1999), provided- and required interfaces are described by separate automata.

The advantage of finite state machines when modelling component-architectures is the availability of efficient checks for interoperability and substitutability. Unfortunately, finite state machines are also among the least powerful models concerning their possibilities modelling complicated protocols. Therefore, attempts have been made, to enhance the power of finite state machines, without loosing their beneficial properties (Reussner, 2001a, Reussner, 2001b).

Architecture definition languages: ADLs have been an active area of research for over 30 years – see (Medvidovic and Taylor, 2000), (Kyaruzi and van Katwijk, 2000), or (Shaw, 2001) for good overviews of the state-of-the-art). While the properties important for architectural system configuration (i.e., interoperability and substitutability checks) have sometimes been a concern during the design of component interface models, research in ADLs has not made use of such approaches for stronger conformance checks. Instead, in languages such as those of (Magee et al., 1995), formal approaches in software architecture have primarily focused on global checks than on local interoperability and substitutability checks. Recently, the Koala language (Kramer et al., 2000) was developed as an extension of Darwin for modelling embedded systems architectures. It enables a more complicated, industrial interface description. Koala's notion of "glue" has some relation to our configuration attributes. However, Koala has yet to incorporate contracts into its notion of connections between components.

7. Conclusions

In this paper, we presented elements of a component-oriented architecture-based design process for modelling and analysing complex distributed software systems. Our methods are centered around the TrustME ADL, which associates contracts to various elements of the architectural design of such software. Our contracts are rooted in the well-known design-by-contract approach for object-oriented program development. However, our approach extends design-by-contract and ADLs considerably. Firstly, the principles of design-by-contract are lifted to deal with components rather than just objects. Secondly, we are dealing with architectural design, significantly higher level than detailed program design. And finally, our methods aim at software qualities to trust in distributed systems:

- protected domains and middleware components;
- well-synchronised and well-behaving software despite its concurrency and distribution with its inherent potential of failure;
- performance and reliability modelling from requirements through to implementation and reuse of COTS components.

Our approach is a true software engineering approach in that it combines

- high-level diagram representations using industrial standards such as UML;
- mathematical specification including state transition systems (for instance specified by finite state machines or Petri nets), logical assertions, and probabilistic quantitative modelling (such as in reliability or performability models);
- execution-based evaluation such as monitoring binary-deployed components against specifications;
- well-defined systematic engineering processes, including modelling, prediction and evaluation.

The paper also examined approaches to CBSE and architecture definition; it used extracts of an electronic commerce reservation system to illustrate selected features of our approach and positioned the latter in the landscape of ADLs and middleware technologies. We also briefly reported on work in progress.

8. References

(van der Aalst et al., 2000) van der Aalst, W., van Hee, K., and van der Toorn, R. (2000). Component-based software architectures: A framework based on inheritance of behaviour. BETA Working Paper Series WP 45, Eindhoven University of Technology.

(Bochmann et al., 1982) Bochmann, G. V., Cerny, E., Gagné, M., Jarda, C., Léveillé, A., Lacaille, C., Maksud, M., Raghunathan, K. S., and Sarikaya, B. (1982). Experience with formal specifications using and extended state transition model. *IEEE Trans. Communications*, 30(12):2506–2511.

(Bosch, 1997) Bosch, J. (1997). Adapting object-oriented components. In Weck, W., Bosch, J., and Szyperski, C., editors, *Proceedings of the Second International Workshop on Component-Oriented Programming (WCOP'97)*, pages 13–22. Turku Centre for Computer Science.

(Bosch, 2001) Bosch, J. (2001). Software Product Lines: Organizational Alternatives. In IEEE, *Proc. 23rd Intl. Conf. Software Engineering, Toronto, Canada, IEEE, 2001*, pp. 91-100.

(Dijkstra, 1969), Dijkstra, E. W., Structured Programming, In Buxton, J. N. and Randell, B. (eds.), *Second NATO Conference on Software Engineering Techniques*, Rome, Italy, NATO Science Committee pp. 84-88

(EJB, 2001) EJB. Sun Microsystems Corp., The Enterprise Java Beans homepage. http:// java.sun.com/products/ejb/.

(Frick et al., 2000) Frick, A., Goos, G., Neumann, R., and Zimmermann, W. (2000). Construction of robust class hierarchies. *Software Practice and Experience*, 30(5):481–543.

(Frick et al., 1996) Frick, A., Zimmer, W., and Zimmermann, W. (1996). Konstruktion robuster und flexibler Klassenbibliotheken. *Informatik, Forschung und Entwicklung*, 11(4):168–178.

(Garlan, 2000) Garlan, D., Software Architecture : A Roadmap, In A. Finkelstein (ed): *The Future of Software Engineering, 22nd Intl Conf. Software Engineering*, IEEE, 2000, pages 91-103

(Han, 2000) Han, J. (2000). Temporal logic based specification of component interaction protocols. In *Proccedings of the 2nd Workshop of Object Interoperability at ECOOP 2000*, Cannes, France.

(Harel, 1987) Harel, D. (1987). Statecharts: a visual approach to complex systems. *Science of Computer Programming*, 8(3):231–274.

(Hoare, 1969) Hoare, C. A. R. (1969). An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580.

(Holzmann, 1991) Holzmann, G. J. (1991). *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, NJ, USA.

(Kleene, 1956) Kleene, S. C. (1956). Representation of events in nerve nets and finite automata. In Shannon, C. and McCarthy, J., editors, *Automata Studies, Annals of Math. Studies 34*, pages 3–40. Princeton, New Jersey.

(Kopetz, 2000) Kopetz H., Software Engineering for Real Time: Roadmap, In A. Finkelstein (ed): *The Future of Software Engineering, 22nd* Intl. Conf. Software Engineering, IEEE, 2000, pages 201-212

(Krämer, 1998) Krämer, B. (1998). Synchronization constraints in object interfaces. In Krämer, B., Papazoglou, M. P., and Schnmidt, H. W., editors, *Information Systems Interoperability*, pages 111–148. Research Studies Press, Taunton, England.

(Krämer and Schmidt, 1987) Krämer, B. and Schmidt, H. W. (1987). Types and modules for net specifications. In Voss, K., Genrich, H. J., and Rozenberg, G., editors, *Concurrency and Nets*, pages 269–286. Springer-Verlag, Berlin - Heidelberg - New York.

(Kramer et al., 2000) Kramer, J., Magee, J., Ng, K., and Dulay, N. (2000). Software architecture description. In *Software A*rchitecture for Product Families: Principles and Practice, pages 31–64. Addison-Wesley.

(Kyaruzi and van Katwijk, 2000) Kyaruzi, J. J. and van Katwijk, J. (2000). Concerns on archtecture-centered software development: A survey. *Transactions of the Society for Design and Process Science*.

(van Lamsweerde, 2000) van Lamsweerde, A., Formal Specification: A Roadmap, In A. Finkelstein (ed): *The Future of Software Engineering, 22nd* Intl. Conf. Software Engineering, IEEE, 2000, pages 201-212

(van Leeuwen, 1990) van Leeuwen, J. (1990). *Formal Models and Semantics, Handbook of Theoretical Computer Science*, volume 2. Elsevier Science Publishers, Amsterdam, The Netherlands.

(Ling et al., 1999) Ling, S., Schmidt, H., and Fletcher, R. (1999). Constructing interoperable components in distributed systems. In *IEEE Proceedings TOOLS Pacific '99, Melbourne*, pages 274–284. IEEE Press.

(Liskov and Wing, 1994) Liskov, B. H. and Wing, J. M. (1994). A behavioural notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841.

(Littlewood and Strigine, 2000) Littlewood, B. and Strigine, L., Software Reliability and Dependibility: A Roadmap, In A. Finkelstein (ed): *The Future of Software Engineering, 22nd* Intl. Conf. Software Engineering, IEEE, 2000

(Lutz, 2000) Lutz, R., Software Engineering for Safety: A Roadmap, In A. Finkelstein (ed): *The Future of Software Engineering, 22nd* Intl. Conf. Software Engineering, IEEE, 2000

(Magee et al., 1995) Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. (1995). Specifying distributed software architectures. *Lecture Notes in Computer Science*, 989:137–155.

(Manna and Pnueli, 1992) Manna, Z. and Pnueli, A. (1992). *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York, USA.

(Medvidovic and Taylor, 2000) Medvidovic, N. and Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. *IEEE* Transactions on Software Engineering, 26(1):70–93.

(Meyer, 1992) Meyer, B. (1992). Applying "design by contract". *IEEE Computer*, 25(10):40–51.

(Meyer, 1997) Meyer, B. (1997a). *Object-Oriented Software Construction*. Prentice/Hall.

(Meyer, Mingins and Schmidt, 1998) Meyer, B., Mingins C. and Schmidt, H., *Providing Trusted Components to the Industry*, IEEE Computer, 5/1998, pp. 104-105

(Milner, 1980) Milner, R. (1980). A calculus of communicating systems. *Lecture Notes in Computer Science*, 92.

(Milner, 1998) Milner, R. (1998). The pi calculus and its applications. In Jaffar, J., editor, *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming (JICSLP*-98), pages 3–4, Cambridge. MIT Press, Cambridge, MA, USA.

(.NET, 2001) .NET. Microsoft Corp., The .net homepage. http://www.microsoft.com/net/default.asp.

(Nierstrasz, 1993) Nierstrasz, O. (1993). Regular types for active objects. In *Proceedings of the 8th ACM* Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-93), volume 28, 10 of *ACM SIGPLAN Notices*, pages 1–15.

(OMG, 2001) Object Management Group (OMG). The CORBA homepage. http://www.corba.org.

(OMG, 1999) Object Management Group (OMG). UML Profile for Enterprise Distributed Object Computing (EDOC) RFP. Technical report, Object Management Group. Available at ftp://ftp.omg.org/pub/docs/ad/99-03-10.pdf.

(OMG, 2000a) Object Management Group (OMG). Meta Object Facility (MOF) Specification. Technical report, Object Management Group.

(OMG, 2000b) Object Management Group (OMG). OMG Unified Modeling Language Specification. Technical report, Object Management Group.

(Petri, 1962) Petri, C. A. (1962). Fundamentals of a theory of asynchronous information flow. In *Information Processing 62*, pages 386–391. IFIP, North-Holland.

(Ran, 2000) Ran, A. (2000). Ares conceptual framework for software architecture. In *Software A*rchitecture for Product Families: Principles and Practice, pages 1–29. Addison-Wesley.

(Reisig, 1985) Reisig, W. (1985). *Petri Nets: An Introduction*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, Germany.

(Reisig, 1992) Reisig, W. (1992). *A Primer in Petri Net Design*. Springer-Verlag, Berlin, Germany.

(Reussner, 2001a) Reussner, R. H. (2001a). Enhanced component interfaces to support dynamic adaption and extension. In *34th Hawaiin International Conference on System Sciences*. IEEE.

(Reussner, 2001b) Reussner, R. H. (2001b). *Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten*. Dissertation, Fakultät für Informatik, Universität Karlsruhe, Germany.

(Reussner, 2001c) Reussner, R. H. (2001c). The use of parameterised contracts for architecting systems with software components. In Weck, W., Bosch, J., and Szyperski, C., editors, *Proceedings of the Sixth International Workshop on Component-Oriented Programming (WCOP'01)*.

(Reussner and Heuzeroth, 1999) Reussner, R. H. and Heuzeroth, D. (1999). A Meta-Protocol and Type system for the Dynamic Coupling of Binary Components. In *Proceedings of the OOPSLA'99 Workshop on Object Oriented Reflection and Software Engineering*.

(de Roever, Langmaack and Pnueli, 1998) de Roever, W.-P., Langmaack, H. and Pnueli, A., editors. *Compositionality: The Significant Difference. International Symposium, COMPOS '97, Bad Malente, Germany, September97. Revised Lectures.* Springer-Verlag, 1998.

(Schmidt, 1998) Schmidt, H. (1998). Compatibility of interoperable objects. In *Information Systems Interoperability*, pages 143–199. Research Studies Press, Taunton, Somerset, England.

(Schmidt, 2001) Schmidt, H. (2001). Trusted Components: Toward Automated Assembly with Predictable Properties. In *Proceedings* ICSE *4th Intl. Workshop Component-Based Sofware Engineering, Toronto, 2001,* IEEE, 2001

(Schmidt and Reussner, 2000) Schmidt, H. and Reussner, R. (2000a). Automatic component adaptation by concurrent state machine retrofitting. Technical Report 2000/81, Monash University, School of Computer Science and Software Engineering. Available from http://www.csse.monash.edu.au/dsse/trustme.

(Schmidt and Zimmermann, 1994) Schmidt H., and Zimmermann, W., A complexity calculus for object-oriented programs, *Journal of Object-Oriented Systems*, pp. 117-147, 1994

(Shaw, 2001) Shaw, M. (2001). The Coming of Age of Software Architecture Research. In IEEE, *Proc. 23rd Intl. Conf. Software Engineering, Toronto, Canada, IEEE, 2001*, pp. 657-664.

(Szyperski, 1998) Szyperski, C. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

(Szyperski, 2000) Szyperski, C. (2000). Components and architecture. *Software Development*, 8 (5).

(Vallecillo et al., 1999) Vallecillo, A., Hernández, J., and Troya, J. (1999). Object interoperability. In Moreira, A. and Demeyer, S., editors, *ECOOP* '99 Reader, number 1743 in LNCS, pages 1–21. Springer-Verlag.

(Yellin and Strom, 1994) Yellin, D. and Strom, R. (1994). Interfaces, Protocols and the Semiautomatic Construction of Software Adaptors. In *Proceedings of the 9th ACM* Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-94), volume 29, 10 of *ACM Sigplan Notices*, pages 176–190.

(Yellin and Strom, 1997) Yellin, D. and Strom, R. (1997). Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333.

(Zaremski and Wing, 1997) Zaremski, A. M. and Wing, J. M. (1997). Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369.